

第 10 章 泛 型

泛型是 JDK1.5 中新加入的元素，它改变了核心 API 中的许多类和方法。使用泛型，可以建立以类型安全模式处理各种数据的类、接口和方法。许多算法不论运用哪一种数据类型，它们在逻辑上是一样的。使用泛型，一旦定义了一个算法，就独立于任何特定的数据类型，而且不需要额外的操作，就可以将这个算法应用到各种数据类型中。正由于泛型的强大功能，从根本上改变了 Java 代码的编写方式。

本章将介绍泛型的语法和应用，同时展示泛型如何提供类型安全。

10.1 泛型的本质

泛型在本质上是指类型参数化。所谓类型参数化，是指用来声明数据的类型本身，也是可以改变的，它由实际参数来决定。在一般情况下，实际参数决定了形式参数的值。而类型参数化，则是实际参数的类型决定了形式参数的类型。

举个简单的例子。方法 `max()` 要求返回两个参数中较大的那个，可以写成：

```
Integer max(Integer a, Integer b){  
    return a>b?a:b;  
}
```

这样编写代码当然没有问题。不过，如果需要比较的不是 `Integer` 类型，而是 `Double` 或是 `Float` 类型，那么就需要另外再写 `max()` 方法。参数有多少种类型，就要写多少个 `max()` 方法。但是无论怎么改变参数的类型，实际上 `max()` 方法体内部的代码并不需要改变。如果有一种机制，能够在编写 `max()` 方法时，不必确定参数 `a` 和 `b` 的数据类型，而等到调用的时候再来确定这两个参数的数据类型，那么只需要编写一个 `max()` 就可以了，这将大大降低程序员编程的工作量。

在 C++ 中，提供了函数模板和类模板来实现这一功能。而从 JDK1.5 开始，也提供了类似的机制：泛型。从形式上看，泛型和 C++ 的模板很相似，但它们是采用完全不同的技术来实现的。

在泛型出现之前，Java 的程序员可以采用一种变通的办法：将参数的类型均声明为 `Object` 类型。由于 `Object` 类是所有类的父类，所以它可以指向任何类对象，但这样做不能保证类型安全。

泛型则弥补了上述做法所缺乏的类型安全，也简化了过程，不必显示地在 `Object` 与实际操作的数据类型之间进行强制转换。通过泛型，所有的强制类型转换都是自动和隐式的。

因此，泛型扩展了重复使用代码的能力，而且既安全又简单。

10.2 一个关于泛型的简单例子

这里用一个简单的例子来开始泛型的学习，让读者对泛型有一个感性的认识。

【例 10.1】 泛型类示例。

//-----文件名 Generic.java, 程序编号 10.1-----

```
//声明一个泛型类
public class Generic<T>{
    T ob; //ob 的类型是 T,现在不能具体确定它的类型,需要到创建对象时才能确定
    Generic(T o){
        ob = o;
    }
    //这个方法的返回类型也是 T
    T getOb(){
        return ob;
    }
    //显示 T 的类型
    void showType(){
        System.out.println("Type of T is:"+ob.getClass().getName() );
    }
}
```

下面这个程序使用上面这个泛型类。

//-----文件名 demoGeneric.java, 程序编号 10.2-----

```
public class demoGeneric{
    public static void main(String args[]){
        //声明一个 Integer 类型的 Generic 变量
        Generic <Integer> iobj;
        //创建一个 Integer 类型的 Generic 对象
        iobj = new Generic<Integer>(100);
        //输出它的一些信息
        iobj.showType();
        int k = iobj.getOb();
        System.out.println("k="+k);
        //声明一个 String 类型的 Generic 变量
        Generic <String> sobj;
        //创建一个 Double 类型的 Generic 对象
        sobj = new Generic<String>("Hello");
        //输出它的一些信息
        sobj.showType();
        String s = sobj.getOb();
        System.out.println("s="+s);
    }
}
```

程序的输出结果如下：

```
Type of T is:java.lang.Integer
k=100
Type of T is:java.lang.String
s=Hello
```

下面来仔细分析一下这个程序。

首先，注意程序是如何声明 **Generic** 的：

```
public class Generic<T>
```

其中，**T** 是类型参数的名称。在创建一个对象时，这个名称用作传递给 **Generic** 的实际类型的占位符。因此，在 **Generic** 中，每当需要类型参数时，就会用到 **T**。注意，**T** 是被括在“<>”中的。每个被声明的类型参数，都要放在尖括号中。由于 **Generic** 使用了类型参数，所以它是一个泛型类，也被称为参数化类型。

然后，**T** 来声明了一个成员变量 **ob**：

```
T ob;
```

由于 **T** 只是一个占位符，所以 **ob** 的实际类型要由创建对象时的参数传递进来。比如，传递给 **T** 的类型是 **String**，那么 **ob** 就是 **String** 类型。

最后，看一下 **Generic** 的构造方法：

```
Generic(T o){
    ob = o;
}
```

它的参数 **o** 的类型也是 **T**。这意味着 **o** 的实际类型，是由创建 **Generic** 对象时传递给 **T** 的类型来决定的。而且，由于参数 **o** 和成员变量 **ob** 都是 **T** 类型，所以无论实际类型是什么，二者都是同一个实际类型。


参数 **T** 还可以用来指定方法的返回类型，如下所示：

```
T getOb(){
    return ob;
}
```

因为 **ob** 是 **T** 类型，所以方法的返回类型必须也由 **T** 来指定。

showType() 方法通过使用 **Class** 对象来获取 **T** 的实际类型，这就是第 9 章介绍的 RTTI 机制。

综合上面的用法，可以看出，**T** 是一个数据类型的说明，它可以用来说明任何实例方法中的局部变量、类的成员变量、方法的形式参数以及方法的返回值。

 **注意：** 类型参数 **T** 不能使用在静态方法中。

程序 9.2 示范了如何使用一个泛型类 **Generic**。它首先声明了 **Generic** 的一个整型版本：

```
Generic <Integer>iobj;
```

其中, 类型 `Integer` 被括在尖括号内, 表明它是一个类型实际参数。在这个整型版本中, 所有对 `T` 的引用都会被替换为 `Integer`。所以 `ob` 和 `o` 都是 `Integer` 类型, 而且方法 `getOb()` 的返回类型也是 `Integer` 类型的。

注意 `Java` 的编译器并不会创建多个不同版本的 `Generic` 类。相反, 编译器会删除所有的泛型信息, 并进行必要的强制类型转换, 这个过程被称为擦拭或擦除。但对程序员而言, 这一过程是透明的, 仿佛编译器创建了一个 `Generic` 的特定版本。这也是 `Java` 的泛型和 `C++` 的模板类在实现上的本质区别。

下面这条语句真正创建了一个 `Integer` 版本的实例对象:

```
iobj = new Generic<Integer>(100);
```

其中, `100` 是普通参数, `Integer` 是类型参数, 它不能被省略。因为 `iobj` 的类型是 `Generic`, 所以用 `new` 返回的引用必须是 `Generic<Integer>` 类型。无论是省略 `Integer`, 还是将其改成其他类型, 都会导致编译错误。例如:

```
iobj = new Generic<Double>(1.234); //错误
```

因为 `iobj` 是 `Generic<Integer>` 类型, 它不能引用 `Generic<Double>` 对象。泛型的一个好处就是类型检查, 所以它能确保类型安全。

再回顾一下 `Generic` 的构造方法的声明:

```
Generic(T o)
```

其中, 实际参数应该是 `Integer` 类型, 而现在的实际参数 `100` 是 `int` 型, 这似乎不正确。实际上, 这里用到了 `Java` 的自动装箱机制 (将在 12.3 节中介绍)。当然, 创建对象也可以写成这种形式:

```
iobj = new Generic(new Integer(100));
```

但这样写没有任何必要。

然后, 程序通过下面的语句获得 `ob` 的值:

```
int k = iobj.getOb();
```

注意, `getOb` 的返回类型也是 `Integer` 的。当它赋值给一个 `int` 变量时, 系统会自动拆箱, 所以没有必要这么来写:

```
int k = iobj.getOb().intValue();
```

后面创建 `String` 版本的过程和前面的完全一样, 在此不再赘述。

最后还有一点需要读者特别注意: 声明一个泛型实例时, 传递给形参的实参必须是类类型, 而不能使用 `int` 或 `char` 之类的简单类型。比如不能这样写:

```
Generic <int> ob = new Generic <int>(100); //错误
```

如果要使用简单类型, 只能使用它们的包装类, 这也是泛型和 `C++` 模板的一个重要区别。

10.3 带两个类型参数的泛型类

在泛型中，可以声明一个以上的类型参数，只需要在这些类型参数之间用逗号隔开。下面看一个简单的例子。

【例 10.2】 带两个类型参数的泛型。

//-----文件名 twoGen.java, 程序编号 10.3-----

```
//本类带有两个类型参数
public class twoGen<T,V>{
    T ob1;
    V ob2;
    //构造方法也可以使用这两个类型参数
    twoGen(T o1, V o2){
        ob1 = o1;
        ob2 = o2;
    }
    //显示 T 和 V 的类型
    void showTypes(){
        System.out.println("Type of T is "+ob1.getClass().getName());
        System.out.println("Type of V is "+ob2.getClass().getName());
    }
    T getOb1(){
        return ob1;
    }
    V getOb2(){
        return ob2;
    }
}
```

下面这个程序演示流如何使用上面这个泛型类。

//-----文件名 simpGen.java, 程序编号 10.4-----

```
public class simpGen{
    public static void main(String args[]){
        twoGen<Integer, String> tgObj; //指定类型参数的实际类型
        //构造方法中需要再次指定类型参数，同时还要传递实际参数
        tgObj = new twoGen<Integer, String>(100,"Hello");
        tgObj.showTypes();
        int v = tgObj.getOb1();
        System.out.println("value: "+v);
        String str = tgObj.getOb2();
        System.out.println("value: "+str);
    }
}
```

程序的输出结果如下：

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 100
value: Hello
```

与只有一个类型参数的泛型相比，本例并没有什么难于理解的地方。Java 并没有规定这两个类型参数是否要相同，比如，下面这样来创建对象也是可以的：

```
twoGen<String, String> tgObj = new twoGen<Integer, String>
("Hello", "World");
```

这样 T 和 V 都是 String 类型，这个例子并没有错。但如果所有的实例都是如此，就没有必要用两个参数。

10.4 有界类型

在前面的例子中，参数类型可以替换成类的任意类型。在一般情况下，这是没有问题的，但有时程序员需要对传递给类型参数的类型加以限制。

比如，程序员需要创建一个泛型类，它包含了一个求数组平均值的方法。这个数组的类型可以是整型、浮点型，但显然不能是字符串类型或是其他非数值类型。如果程序员写出如下所示的泛型类。

//-----文件名 Stats.java，程序编号 10.5-----

```
class Stats<T>{
    T [] nums;
    Stats (T [] obj){
        nums = obj;
    }
    double average(){
        double sum = 0.0;
        for (int i=0; i<nums.length; ++i)
            sum += nums[i].doubleValue();    //这里有错误!
        return sum / nums.length;
    }
}
```

其中，nums[i].doubleValue()是返回 Integer、Double 等数据封装类转换成双精度数后的值，所有的 Number 类的子类都有这个方法。但问题是，编译器无法预先知道，程序员的意图是只能使用 Number 类来创建 Stats 对象，因此，编译时会报告找不到 doubleValue() 方法。

为了解决上述问题，Java 提供了有界类型（bounded types）。在指定一个类型参数时，可以指定一个上界，声明所有的实际类型都必须是这个超类的直接或间接子类。语法形式如下：

```
class classname <T extends superclass>
```

采用这种方法，可以正确地编写 Stats 类。

【例 10.3】 有界类型程序示例。

//-----文件名 Stats.java, 程序编号 10.6-----

```
//下面这个泛型的实际类型参数只能是 Number 或它的子类
class Stats<T extends Number>{
    T [] nums;
    Stats (T [] obj){
        nums = obj;
    }
    double average(){
        double sum = 0.0;
        for (int i=0; i<nums.length; ++i)
            sum += nums[i].doubleValue();    //现在正确!
        return sum / nums.length;
    }
}
```

程序 10.7 演示了如何使用这个类。

//-----文件名 demoBounds.java, 程序编号 10.7-----

```
public class demoBounds{
    public static void main(String args[]){
        Integer inums[] = {1,2,3,4,5};
        Stats <Integer> iobj = new Stats<Integer>(inums);
        System.out.println("平均值为: "+iobj.average());
        Double dnums[] = {1.1,2.2,3.3,4.4,5.5};
        Stats <Double> dobj = new Stats<Double>(dnums);
        System.out.println("平均值为: "+dobj.average());
        //如果像下面这样创建 String 类型的对象将无法编译通过
        //String snums[] = {"1","2","3","4","5"};
        //Stats <String> sobj = new Stats<String>(snums);
        //System.out.println("平均值为: "+sobj.average());
    }
}
```


程序的输出结果如下：

平均值为: 3.0

平均值为: 3.3


程序 10.6 和程序 10.7 的上界都是类，实际上，接口也可以用来做上界。比如：

```
class Stats<T extends Comparable>
```

 **注意：** 这里使用的关键字仍然是 extends 而非 implements。

一个类型参数可以有多个限界，比如：

```
class Stats<T extends Comparable & Serializable>
```

 **注意：**限界类型用“&”分隔，因为逗号用来分隔类型参数。在多个限界中，可以有多个接口，但最多只能有一个类。如果用一个类作为限界，它必须是限界列表中的第一个。

10.5 通配符参数

前面介绍的泛型已经可以解决大多数的实际问题，但在某些特殊情况下，仍然会有一些问题无法轻松地解决。

以 `Stats` 类为例，假设在其中存在一个名为 `doSomething()` 的方法，这个方法有一个形式参数，也是 `Stats` 类型，如下所示：

```
class Stats<T>{
    .....
    void doSomething(Stats <T> ob){
        System.out.println(ob.getClass().getName());
    }
}
```

如果在使用的時候，像下面这样写是有问题的：

```
Integer inums[] = {1,2,3,4,5};
Stats <Integer> iobj = new Stats<Integer>(inums);
Double dnums[] = {1.1,2.2,3.3,4.4,5.5};
Stats <Double> dobj = new Stats<Double>(dnums);
dobj.doSomething(iobj); //错误
```

注意看出现错误的这条语句：

```
dobj.doSomething(iobj);
```

`dobj` 是 `Stats<Double>` 类型，`iobj` 是 `Stats<Integer>` 类型，由于实际类型不同，而声明时用的是：

```
void doSomething(Stats <T> ob)
```

它的类型参数也是 `T`，与声明对象时的类型参数 `T` 相同。于是在实际使用中，就要求 `iobj` 和 `dobj` 的类型必须相同。

读者也许会想，将 `doSomething` 的声明改一下：

```
void doSomething(Stats <U> ob)
```

但这样是无法通过编译的，因为并不存在一个 `State<U>` 的泛型类。解决这个问题的办法是使用 `Java` 提供的通配符“`?`”，它的使用形式如下：

```
genericClassName <?>
```

比如，上面的 `doSomething` 可以声明成这个样子：


```
void doSomething(Stats <?> ob)
```

它表示这个参数 `ob` 可以是任意的 `Stats` 类型，于是调用该方法的对象就不必和实际参数对象类型一致了。下面这个例子实际演示了通配符的使用。

【例 10.4】 通配符使用示例。

//-----文件名 Stats.java, 程序编号 10.8-----

```
class Stats<T extends Number>{
    T [] nums;
    Stats (T [] obj){
        nums = obj;
    }
    double average(){
        double sum = 0.0;
        for (int i=0; i<nums.length; ++i)
            sum += nums[i].doubleValue();
        return sum / nums.length;
    }
    void doSomething(Stats <?> ob){ //这里使用了类型通配符
        System.out.println(ob.getClass().getName());
    }
}
```

然后如程序 10.9 所示来调用它：

//-----文件名 demoWildcard.java, 程序编号 10.9-----

```
public class demoWildcard{
    public static void main(String args[]){
        Integer inums[] = {1,2,3,4,5};
        Stats <Integer> iobj = new Stats<Integer>(inums);
        Double dnums[] = {1.1,2.2,3.3,4.4,5.5};
        Stats <Double> dobj = new Stats<Double>(dnums);
        dobj.doSomething(iobj); //iobj 和 dobj 的类型不相同
    }
}
```

程序的输出结果如下：

```
Stats
```

读者应该注意到这个声明：

```
void doSomething(Stats <?> ob) //这里使用了类型通配符
```

它与泛型类的声明有区别，泛型类的声明中，`T` 是有上界的：

```
class Stats<T extends Number>
```

其中，通配符“?”有一个默认的上界，就是 `Number`。如果想改变这个上界，也是可以的，比如：

```
Stats <? extends Integer> ob
```

但是不能写成这样：

```
Stats <? extends String> ob
```

因为 `Integer` 是 `Number` 的子类，而 `String` 不是 `Number` 的子类。通配符无法将上界改变得超出泛型类声明时的上界范围。

最后读者需要注意一点，通配符是用来声明一个泛型类的变量的，而不能创建一个泛型类。比如下面这种写法是错误的：

```
class Stats<? extends Number>{.....}
```

10.6 泛型方法

在 C++ 中，除了可以创建模板类，还可以创建模板函数。在 Java 中也提供了类似的功能：泛型方法。一个方法如果被声明成泛型方法，那么它将拥有一个或多个类型参数，不过与泛型类不同，这些类型参数只能在它所修饰的泛型方法中使用。

创建一个泛型方法常用的形式如下：

```
[访问权限修饰符] [static] [final] <类型参数列表> 返回值类型 方法名([形式参数列表])
```

- ❑ 访问权限修饰符（包括 `private`、`public`、`protected`）、`static` 和 `final` 都必须写在类型参数列表的前面。
- ❑ 返回值类型必须写在类型参数表的后面。
- ❑ 泛型方法可以写在一个泛型类中，也可以写在一个普通类中。由于在泛型类中的任何方法，本质上都是泛型方法，所以在实际使用中，很少会在泛型类中再用上面的形式来定义泛型方法。
- ❑ 类型参数可以用在方法体中修饰局部变量，也可以用在方法的参数表中，修饰形式参数。
- ❑ 泛型方法可以是实例方法或是静态方法。类型参数可以使用在静态方法中，这是与泛型类的重要区别。

使用一个泛型方法通常有两种形式：

```
<对象名|类名>.<实际类型>方法名(实际参数表);
```

和：

```
[对象名|类名].方法名(实际参数表);
```

如果泛型方法是实例方法，要使用对象名作为前缀。如果是静态方法，则可以使用对象名或类名作为前缀。如果是在类的内部调用，且采用第二种形式，则前缀都可以省略。

注意到这两种调用方法的差别在于前面是否显示地指定了实际类型。是否要使用实际类型，需要根据泛型方法的声明形式以及调用时的实际情况（就是看编译器能否从实际参数表中获得足够的类型信息）来决定。下面来看一个例子。

【例 10.5】 泛型方法使用示例。

//-----文件名 demoGenMethods.java, 程序编号 10.10-----

```

public class demoGenMethods{
    //定义泛型方法，有一个形式参数用类型参数 T 来定义
    public static <T> void showGenMsg(T ob, int n){
        T localOb = ob; //局部变量也可以用类型参数 T 来定义
        System.out.println(localOb.getClass().getName());
    }
    public static <T> void showGenMsg(T ob){
        System.out.println(ob.getClass().getName());
    }
    public static void main(String args[]){
        String str = "parameter";
        Integer k = new Integer(123);
        //用两种不同的方法调用泛型方法
        demoGenMethods.<Integer>showGenMsg(k,1);
        showGenMsg(str);
    }
}

```

程序中定义的两个泛型方法都是静态方法，这在泛型类中是不允许的。而且这两个泛型方法相互重载（参数的个数不同）。在方法体中，类型参数 T 的使用和泛型类中的使用是相同的。

再来看看如何调用这两个泛型方法：

```

demoGenMethods.<Integer>showGenMsg(k,1);
showGenMsg(str);

```

在第一种调用形式中，传入了一个实际类型：<Integer>，它表明类型参数是 Integer 类型。要注意它的写法，在这种情况下，不能省略作为前缀的类名，也就是不能写成这样：

```

<Integer>showGenMsg(k,1);

```

由于传递了一个实际的类型参数 Integer，所以编译器知道如何将方法内部的占位符 T 替换掉。不过需要注意，实际参数 k 的类型必须也是 Integer 型，否则编译器会报错。

第二种调用形式明显要简洁一些：

```

showGenMsg(str);

```

由于实参 str 是 String 类型的，编译器已经有了足够多的信息知道类型参数 T 是 String 类型。程序的输出结果如下：

```

java.lang.Integer
java.lang.String

```

由于两种形式都能完成任务，而第二种明显要比第一种方便，所以多数情况下会使用第二种方式。不过在某些情况下，实参无法提供足够的类型信息给编译器，那么就需要使用第一种形式。例如：

```
public <T> void doSomething(){
    T ob;
    .....
}
```

调用它的时候，根本就没有实际参数，所以编译器无法知道 `T` 的实际类型，这种情况下，就必须使用第一种形式。

前面还提到，泛型方法也可以写在泛型类中，比如：

```
public class Generic<T>{
    public <U> void showGenMsg(U ob){
        System.out.println(ob.getClass().getName());
    }
    .....
}
```

这样写当然没有错误，但多数程序员都会将这个泛型方法所需要的类型参数 `U` 写到类的头部，即让泛型类带两个参数：

```
public class Generic<T, U>{
    public void showGenMsg(U ob){
        System.out.println(ob.getClass().getName());
    }
    .....
}
```

这样写，类的结构更为清晰。只有一种情况下必须在泛型类中再将方法声明为泛型方法：方法本身是静态的，那就无法像上面那样更改了。

10.7 泛型接口

除了泛型类和泛型方法，还可以使用泛型接口。泛型接口的定义与泛型类非常相似，它的声明形式如下：

```
interface 接口名<类型参数表>
```

下面的例子创建了一个名为 `MinMax` 的接口，用来返回某个对象集的最小值或最大值。

【例 10.6】 泛型接口示例。

//-----文件名 `MinMax.java`，程序编号 10.11-----

```
interface MinMax<T extends Comparable<T>>{
    T min();
    T max();
}
```

这个接口没有什么特别难懂的地方，类型参数 `T` 是有界类型，它必须是 `Comparable` 的子类。注意到 `Comparable` 本身也是一个泛型类，它是由系统定义在类库中的，可以用来

比较两个对象的大小。

接下来的事情是实现这个接口，这需要定义一个类来实现。笔者实现的版本如下：

//-----文件名 MyClass.java, 程序编号 10.12-----

```
class MyClass<T extends Comparable<T>> implements MinMax<T>{
    T [] vals;
    MyClass(T [] ob){
        vals = ob;
    }
    public T min(){
        T val = vals[0];
        for(int i=1; i<vals.length; ++i)
            if (vals[i].compareTo(val) < 0)
                val = vals[i];
        return val;
    }
    public T max(){
        T val = vals[0];
        for(int i=1; i<vals.length; ++i)
            if (vals[i].compareTo(val) > 0)
                val = vals[i];
        return val;
    }
}
```

类的内部并不难懂，只要注意 **MyClass** 的声明部分：

```
class MyClass<T extends Comparable<T>> implements MinMax<T>
```

看上去有点奇怪，它的类型参数 **T** 必须和要实现的接口中的声明完全一样。反而是接口 **MinMax** 的类型参数 **T** 最初是写成有界形式的，现在已经不再需要重写一遍。如果重写成下面这个样子：

```
class MyClass<T extends Comparable<T>> implements MinMax<T extends
Comparable<T>>
```

编译将无法通过。

通常，如果一个类实现了一个泛型接口，则此类也是泛型类。否则，它无法接受传递给接口的类型参数。比如，下面这种声明是错误的：

```
class MyClass implements MinMax<T>
```

因为在类 **MyClass** 中需要使用类型参数 **T**，而类的使用者无法把它的实际参数传递进来，所以编译器会报错。不过，如果实现的是泛型接口的特定类型，比如：

```
class MyClass implements MinMax<Integer>
```

这样写是正确的，现在这个类不再是泛型类。编译器会在编译此类时，将类型参数 **T** 用 **Integer** 代替，而无需等到创建对象时再处理。

最后写一个程序来测试 **MyClass** 的工作情况。

//-----文件名 demoGenIF.java, 程序编号 10.13-----

```
public class demoGenIF{
    public static void main(String args[]){
        Integer inums[] = {56,47,23,45,85,12,55};
        Character chs[] = {'x','w','z','y','b','o','p'};
        MyClass<Integer> iob = new MyClass<Integer>(inums);
        MyClass<Character> cob = new MyClass<Character>(chs);
        System.out.println("Max value in inums: "+iob.max());
        System.out.println("Min value in inums: "+iob.min());
        System.out.println("Max value in chs: "+cob.max());
        System.out.println("Min value in chs: "+cob.min());
    }
}
```

在使用类 `MyClass` 创建对象的方式上, 和前面使用普通的泛型类没有任何区别。程序的输出结果如下:

```
Max value in inums: 85
Min value in inums: 12
Max value in chs: z
Min value in chs: b
```

10.8 泛型类的继承

和普通类一样, 泛型类也是可以继承的, 任何一个泛型类都可以作为父类或子类。不过泛型类与非泛型类在继承时的主要区别在于: 泛型类的子类必须将泛型父类所需要的类型参数, 沿着继承链向上传递。这与构造方法参数必须沿着继承链向上传递的方式类似。

10.8.1 以泛型类为父类

当一个类的父类是泛型类时, 这个子类必须要把类型参数传递给父类, 所以这个子类也必定是泛型类。下面是一个简单的例子。

【例 10.7】 继承泛型类示例。

//-----文件名 superGen.java, 程序编号 10.14-----

```
public class superGen<T> { //定义一个泛型类
    T ob;
    public superGen(T ob){
        this.ob = ob;
    }
    public superGen(){
        ob = null;
    }
}
```

```

public T getOb(){
    return ob;
}
}

```

接下来定义它的一个子类：

//-----文件名 derivedGen.java, 程序编号 10.15-----

```

public class derivedGen <T> extends superGen<T>{
    public derivedGen(T ob){
        super(ob);
    }
}

```

注意 derivedGen 是如何声明成 superGen 的子类的：

```

public class derivedGen <T> extends superGen<T>

```

这两个类型参数必须用相同的标识符 T。这意味着传递给 derivedGen 的实际类型也会传递给 superGen。例如，下面的定义：

```

derivedGen<Integer> number = new derivedGen<Integer>(100);

```

将 Integer 作为类型参数传递给 derivedGen，再由它传递给 superGen，因此，后者的成员 ob 也是 Integer 类型。

虽然 derivedGen 里面并没有使用类型参数 T，但由于它要传递类型参数给父类，所以它不能定义成非泛型类。当然，derivedGen 中也可以使用 T，还可以增加自己需要的类型参数。下面这个程序展示了一个更为复杂的 derivedGen 类。

//-----文件名 derivedGen.java, 程序编号 10.16-----

```

public class derivedGen <T, U> extends superGen<T>{
    U dob;
    public derivedGen(T ob1, U ob2){
        super(ob1);    //传递参数给父类
        dob = ob2;    //为自己的成员赋值
    }
    public U getDob(){
        return dob;
    }
}

```

使用泛型子类和使用其他的泛型类没有区别，使用者根本无需知道它是否继承了其他的类。下面是一个测试用的程序：

//-----文件名 demoHerit_1.java, 程序编号 10.17-----

```

public class demoHerit_1{
    public static void main(String args[]){
        //创建子类的对象，它需要传递两个参数，Integer 类型给父类，自己使用 String 类型
    }
}

```

```

        derivedGen<Integer,String> oa=new derivedGen<Integer,String>
        (100,"Value is: ");
        System.out.print(oa.getDob());
        System.out.println(oa.getOb());
    }
}

```

程序的输出结果如下：

```
Value is: 100
```

10.8.2 以非泛型类为父类

前面介绍的泛型类是以泛型类作为父类，一个泛型类也可以以非泛型类为父类。此时，不需要传递类型参数给父类，所有的类型参数都是为自己准备的。下面是一个简单的例子。

【例 10.8】 继承非泛型类示例。

//-----文件名 nonGen.java，程序编号 10.18-----

```

public
}

```

接下来定义一个泛型类作为它的子类：

//-----文件名 derivedNonGen.java，程序编号 10.19-----

```

public class derivedNonGen<T> extends nonGen{
    T ob;
    public derivedNonGen(T ob, int n){
        super(n);
        this.ob = ob;
    }
    public T getOb(){
        return ob;
    }
}

```

这个泛型类仍然传递了一个普通参数给它的父类，所以它的构造方法需要两个参数。下面是用于测试的程序：

//-----文件名 demoHerit_2.java，程序编号 10.20-----

```

public class demoHerit_2{
    public static void main(String args[]){
        derivedNonGen<String> oa =new derivedNonGen<String> ("Value is: ",
        100);
        System.out.print(oa.getOb());
        System.out.println(oa.getNum());
    }
}

```


程序的输出结果如下：

```
Value is: 100
```

10.8.3 运行时类型识别

和其他的非泛型类一样，泛型类也可以进行运行时类型识别的操作，既可以使用反射机制，也可以采用传统的方法。比如，`instanceof` 操作符。


需要注意的是，由于在 JVM 中，泛型类的对象总是一个特定的类型，此时，它不再是泛型。所以，所有的类型查询都只会产生原始类型，无论是 `getClass()` 方法，还是 `instanceof` 操作符。

例如，对象 `a` 是 `Generic<Integer>` 类型（`Generic` 是例 10.1 中定义的泛型类），那么

```
a instanceof Generic<? >
```

测试结果为真，下面的测试结果也为真：

```
a instanceof Generic
```

 **注意：**尖括号中只能写通配符“`?`”，而不能写 `Integer` 之类确定的类型。实际上在测试时，“`?`”会被忽略。

同样道理，`getClass()` 返回的也是原始类型。若 `b` 是 `Generic<String>` 类型，下面的语句：

```
a.getClass() == b.getClass()
```

测试结果也为真。下面的程序演示了这些情况。

【例 10.9】 泛型类的类型识别示例 1。

//-----文件名 demoRTTI_1.java，程序编号 10.21-----

```
public class demoRTTI_1{
    public static void main(String args[]){
        Generic<Integer> iob = new Generic<Integer>(100);
        Generic<String> sob = new Generic<String>("Good");
        if (iob instanceof Generic)
            System.out.println("Generic<Integer> object is instance of Generic");
        if (iob instanceof Generic<?>)
            System.out.println("Generic<Integer> object is instance of
            Generic<?>");
        if (iob.getClass() == sob.getClass())
            System.out.println("Generic<Integer> class equals Generic<String>
            class");
    }
}
```

程序的输出结果如下：

```
Generic<Integer> object is instance of Generic
```

```
Generic<Integer> object is instance of Generic<?>
Generic<Integer> class equals Generic<String> class
```

泛型类对象的类型识别还有另外一个隐含的问题，它会在继承中显示出来。例如，对象 **a** 是某泛型子类的对象，当用 `instanceof` 来测试它是否为父类对象时，测试结果也为真。

下面的例子使用了例 10.7 中的两个类：`superGen` 和 `derivedGen`。

【例 10.10】 泛型类的类型识别示例 2。

//-----文件名 demoRTTI_2.java, 程序编号 10.22-----

```
public class demoRTTI_2{
    public static void main(String args[]){
        superGen <Integer> oa = new superGen<Integer>(100);
        derivedGen<Integer,String> ob = new derivedGen<Integer,
        String>(200,"Good");
        if (oa instanceof derivedGen)
            System.out.println("superGen object is instance of derivedGen");
        if (ob instanceof superGen)
            System.out.println("derivedGen object is instance of superGen");
        if(oa.getClass() == ob.getClass())
            System.out.println("superGen class equals derivedGen class");
    }
}
```

程序的输出结果如下：

```
derivedGen object is instance of superGen
```

从上述结果中可以看出，只有子类对象被 `instanceof` 识别为父类对象。

10.8.4 强制类型转换

和普通对象一样，泛型类的对象也可以采用强制类型转换转换成另外的泛型类型，不过只有当两者在各个方面兼容时才能这么做。

泛型类的强制类型转换的一般格式如下：

```
(泛型类名<实际参数>) 泛型对象
```

下面的例子展示了两个转换，一个是正确的，一个是错误的。它使用了例 10.7 中的两个类：`superGen` 和 `derivedGen`。

【例 10.11】 强制类型转换示例。

//-----文件名 demoForceChange.java, 程序编号 10.23-----

```
public class demoForceChange{
    public static void main(String args[]){
        superGen <Integer> oa = new superGen<Integer>(100);
        derivedGen<Integer,String> ob = new derivedGen<Integer, String>
        (200,"Good");
        //试图将子类对象转换成父类，正确
        if ((superGen<Integer>)ob instanceof superGen)
```

```

        System.out.println("derivedGen object is changed to superGen");
        // 试图将父类对象转换成子类，错误
        if ((derivedGen<Integer,String>)oa instanceof derivedGen)
            System.out.println("superGen object is changed to derivedGen");
    }
}

```

这个程序编译时会出现一个警告，如果不理会这个警告，继续运行程序，会得到下面的结果：

```


derivedGen object is changed to superGen
Exception in thread "main" java.lang.ClassCastException: superGen
    at demoForceChange.main(demoForceChange.java:7)

```

第一个类型转换成功，而第二个则不能成功。因为 `oa` 转换成子类对象时，无法提供足够的类型参数。由于强制类型转换容易引起错误，所以对于泛型类的强制类型转换的限制是很严格的，即便是下面这样的转换，也不能成功：

```
(derivedGen<Double,String>)ob
```

因为 `ob` 本身的第一个实际类型参数是 `Integer` 类型，无法转换成 `Double` 类型。

 **提示：**建议读者如果不是十分必要，不要做强制类型转换的操作。

10.8.5 继承规则

现在再来讨论一下关于泛型类的继承规则。前面所看到的泛型类之间是通过关键字 `extends` 来直接继承的，这种继承关系十分的明显。不过，如果类型参数之间具有继承关系，那么对应的泛型是否也会具有相同的继承关系呢？比如，`Integer` 是 `Number` 的子类，那么 `Generic<Integer>` 是否是 `Generic<Number>` 的子类呢？答案是：否。比如，下面的代码将不会编译成功：

```
Generic<Number> oa = new Generic<Integer>(100);
```

因为 `oa` 的类型不是 `Generic<Integer>` 的父类，所以这条语句无法编译通过。事实上，无论类型参数之间是否存在联系，对应的泛型类之间都是不存在联系的。如图 10.1 所示。

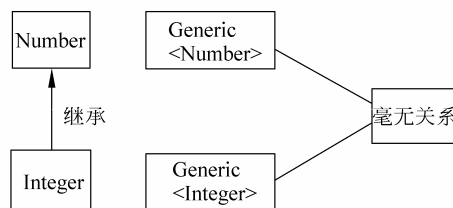


图 10.1 Generic 类之间没有继承关系

这一限制看起来过于严格，但对于类型安全而言是非常必要的。

10.9 擦 拭

通常，程序员不必知道有关 Java 编译器将源代码转换成为 class 文件的细节。但在使用泛型时，对此过程进行一般的了解是必要的，因为只有了解这一细节，程序员才能理解泛型的工作原理，以及一些令人惊讶的行为——如果程序员不知道，可能会认为这是错误。

Java 在 JDK1.5 以前的版本中是没有泛型的，为了保证对以前版本的兼容，Java 采用了与 C++ 的模板完全不同的方式来处理泛型（尽管它们二者的使用看上去很相似），Java 采用的方法称为擦拭。

擦拭的工作原理是这样的：当 Java 代码被编译时，全部泛型类型的信息会被删除（擦拭）。也就是使用类型参数来了替换它们的限界类型，如果没有指定界限，则默认类型是 Object，然后运用相应的强制转换（由类型参数来决定）以维持与类型参数的类型兼容。编译器会强制这种类型兼容。对于泛型来说，这种方法意味着在运行时不存在类型参数，它们仅仅只是一种源代码机制。

为了更好地理解泛型是如何工作的，请看下面两个例子。

【例 10.12】 无限界的擦拭。

//-----文件名 Gen.java，程序编号 10.24-----

```
//默认情况下，T 是由 Object 限界
public class Gen<T>{
    //下面所有的 T 将被 Object 所代替
    T ob;
    Gen(T ob){
        this.ob = ob;
    }
    T getOb(){
        return ob;
    }
}
```

将这个类编译完成后，在命令行输入：

```
javap Gen
```

javap 是由系统提供的一个反编译命令，可以获取 class 文件中的信息或者是反汇编代码。该命令执行后，输出结果如下：

```
Compiled from "Gen.java"
public class Gen extends java.lang.Object{
    java.lang.Object ob;
    Gen(java.lang.Object);
    java.lang.Object getOb();
}
```

```
}
```

从上述结果中可以看出，所有被 T 占据的位置都被 java.lang.Object 所取代，这也是前面将 T 称为“占位符”的原因。

如果类型参数指定了上界，那么就会用上界类型来代替它，下面的例子表明了这一点。

【例 10.13】 有限界的擦拭。

//-----文件名 GenStr.java，程序编号 10.25-----

```
//T 是由 String 限界
public class GenStr<T extends String>{
    //下面所有的 T 将被 String 所代替
    T ob;
    GenStr(T ob){
        this.ob = ob;
    }
    T getOb(){
        return ob;
    }
}
```

用 javap 来反编译这个类，可以得到下面的结果：

```
Compiled from "GenStr.java"
public class GenStr extends java.lang.Object{
    java.lang.String ob;
    GenStr(java.lang.String);
    java.lang.String getOb();
}
```

在使用泛型对象时，实际上所有的类型信息也都会被擦拭，编译器自动插入强制类型转换。比如：

```
Gen<Integer> oa = new Gen<Integer>(100);
Gen<Integer> ob = oa.getOb();
```

由于 getOb 的实际返回类型是 Object 类型，所以后面这一句相当于：

```
Gen<Integer> ob = (Gen<Integer>)oa.getOb();
```

正是由于擦拭会去除实际的类型，所以，在运行时做类型识别将得到原始类型，而非具体指定的参数类型。这一点在 10.8.3 小节中已经详细介绍过了。

10.10 擦拭带来的错误

擦拭是一种很巧妙的办法，但它有时候会带来一些意想不到的错误：两个看上去并不相同的泛型类或是泛型方法，由于擦拭的作用，最后会得到相同的类和方法。这种错误，也被称为冲突。冲突主要发生在下述三种情况。

10.10.1 静态成员共享问题

在泛型类中可以有静态的属性或者方法。前面已经介绍过，静态方法不能使用类型参数。那么，其中的静态成员是否可以使用类型参数或者是本泛型类的对象呢？答案是：否。下面的例子展示了这一错误。

【例 10.14】 静态成员不能使用类型参数。

//-----文件名 foo.java，程序编号 10.26-----

```
public class foo<T>{
    static T sa; //错误
    static foo<T> sb = new foo<T>(); //错误
    static foo<Integer> si = new foo<Integer>(100);
    static foo<String> ss = new foo<String>("Good");
    T ob;
    foo( T ob){
        this.ob = ob;
    }
    foo(){
        this.ob = null;
    }
}
```

出现错误的两个变量 `sa` 和 `sb` 都采用不同的形式使用了类型参数 `T`。由于它们是静态成员，是独立于任何对象的，也可以在对象创建之前就被使用。此时，编译器无法知道用哪一个具体的类型来替代 `T`，所以编译器不允许这样使用。在静态方法中不允许出现类型参数 `T` 也是出于同样的道理。

10.10.2 重载冲突问题

擦拭带来的另外一个问题是重载的冲突，向下面这样的两个方法重载：

```
void conflict(T o){ }
void conflict(Object o){ }
```

由于在编译时，`T` 会被 `Object` 所取代，所以这两个实际上声明的是同一个方法，重载就出错了。另一种情形不是很直观，比如下面的方法重载：

```
public int conflict(foo<Integer> i){}
public int conflict(foo<String> s){}
```

编译时会报错：

```
名称冲突： conflict(foo<java.lang.Integer>) 和 conflict(foo<java.
lang.String>) 具有相同疑符
```

注意到编译器只是怀疑它可能会引发冲突，如果加上一些其他信息能够消除这一歧，编译是可以通过的。比如，这样写：

```
public int conflict(foo<Integer> i){}
public String conflict(foo<String> s){}
```

只是将返回类型修改一下，编译器就能从调用者处获得足够的信息，编译可以通过。

10.10.3 接口实现问题

由于接口也可以是泛型接口，而一个类又可以实现多个泛型接口，所以也可能会引发冲突。比如：

```
class foo implements Comparable<Integer>, Comparable<Long>
```

由于 `Comparable<Integer>`、`Comparable<Long>` 都被擦除成 `Comparable`，所以这实际上是实现的同一个接口。要实现泛型接口，只能实现具有不同擦除效果的接口。否则，只能按照 10.7 节所介绍的来写：

```
class foo<T> implements Comparable<T>
```

10.11 泛型的局限

使用泛型时有一些限制，多数限制是由于类型擦除引起的。这些局限主要包括：

10.11.1 不能使用基本类型

泛型中使用的所有类型参数都是类类型，不能使用基本类型。比如，可以用 `Generic<Integer>`，而不能用 `Generic<int>`。原因很简单，基本类型无法用 `Object` 来替换。

尽管这有点令人（特别是 C++ 程序员）感到麻烦，不过并不是什么大问题。因为 Java 中只有 8 个基本类型，而且系统为每个基本类型都提供了包装类。即便这些包装类不能完成预定的任务，也完全可以使用独立的类和方法来处理它们。

10.11.2 不能使用泛型类异常

Java 中不能抛出也不能捕获泛型类的异常。事实上，泛型类继承 `Throwable` 及其子类都不合法，例如下面的定义将不会通过编译：

```
class MyException<t> extends Exception{.....}
```

也不能在 `catch` 子句中使用参数类型。例如，下面的方法不能通过编译：

```
public void doSomething(T oa){
    try{
        throw a;    //错误
    }catch(T el){ //错误
    }
```

```
.....
}
}
```

先来看第一个错误，抛出一个 `T` 类型的对象 `oa` 作为异常，这是不允许的。因为在没指定上界的情况下，`T` 会被擦拭成 `Object` 类，而 `Object` 类显然不会是 `Throwable` 的子类，因此它不符合异常的有关规定。第二个错误的原因也是一样的。

改正第一个问题的办法是在类的头部加上限界：

```
<T extends Throwable>
```

但是没有什么办法能够改正第二个错误。编译器在处理 `catch` 语句时，将它当作静态上下文来看待，尽管这么做给程序员带来了一点不便。但考虑到 `catch` 语句必须在异常发生时才会执行，而且必须有足够的运行时信息，而泛型在这一方面不如非泛型类，所以这么做仍然是可以接受的。

10.11.3 不能使用泛型数组

Java 规定不能使用泛型类型的数组，比如：

```
Generic<Integer> arr[] = new Generic<Integer>[10];
```

在擦拭之后，`arr` 的类型为 `Generic[]`，这里可以将它转换成为 `Object[]` 数组：

```
Object [] obj = arr;
```

数组可以记住它的元素类型，如果试图存入一个错误类型的元素，编译器就会抛出一个 `ArrayStoreException` 类型的异常。比如：

```
obj[0]="foolish"; //抛出异常
```

不过，对于泛型而言，擦拭将降低这一机制的效率。像下面这样的赋值：

```
obj[0]=new Generic<String>("foolish");
```

做了擦拭之后，只剩下 `Generic`，编译器将无法检测到 `String` 和原始定义中 `Integer` 的不兼容，所以可以通过数组存储的检测。但在运行时会导致类型错误。所以，禁止使用泛型数组。

10.11.4 不能实例化参数类型对象

不能直接使用参数类型来构造一个对象。比如，下面这种写法是错误的：

```
public class foo< T >{
    T ob = new T(); //错误
}
```

这里的 `T` 擦拭成 `Object`，而程序员的本意肯定不是希望调用 `new Object()`。

类似的，也不能创建一个泛型数组：

```
public class foo< T >{  
    T [] ob = new T [100]; //错误  
}
```

因为它实际上是创建的数组 `Object[100]`。

通常情况下，上面这些由参数类型所指定的对象和数组都不会在泛型类中创建，而是由外部创建泛型对象时传递进来的。如果一定要在泛型类中创建参数类型所指定的对象和数组，可以通过反射机制中的 `Class.newInstance()`和 `Array.newInstance()`方法。

10.12 本章小结

本章全面介绍了泛型的定义和使用。泛型是 **JDK1.5** 中参照 **C++**模板所新增的类。作为一种功能强大的新型类，它为程序员编程提供了很大的便利，降低了程序员重复编写逻辑相同代码的工作量，但同时也增加了出错的可能，所以使用的时候一定要慎重。

Java 泛型的设计经过了 5 年左右的时间才定型，它不仅功能强大，而且使用也比较方便，能够最大限度地提供类型安全检测。其中，有界类型、通配符是体现这一思想的有力武器。

在大多数情况下，泛型被设计用来处理集合。实际上，**JDK** 自己提供的 `ArrayList` 就是一个泛型类。大多数应用程序员只要熟练使用系统提供的泛型类，就足够应付大多数的程序需要了。